Certifying Top-Down Decision-DNNF Compilers

Florent Capelli, Jean-Marie Lagniez, Pierre Marquis

Certifying Top-Down Decision-DNNF Compilers

Florent Capelli,[1] **Jean-Marie Lagniez,**[2] **Pierre Marquis**[2, 3]
[1] Université de Lille & CNRS & Inria & UMR 9189 - CRIStAL
[2] CRIL, Université d'Artois & CNRS
[3] Institut Universitaire de France
florent.capelli@univ-lille.fr, lagniez@cril.fr, marquis@cril.fr

Abstract

Certifying the output of tools solving complex problems so as to ensure the correctness of the results they provide is of tremendous importance. Despite being widespread for SAT-solvers, this level of exigence has not yet percolated for tools solving more complex tasks, such as model counting or knowledge compilation. In this paper, the focus is laid on a general family of top-down Decision-DNNF compilers. We explain how those compilers can be tweaked so as to output certifiable Decision-DNNF circuits, which are mainly standard Decision-DNNF circuits decorated by annotations serving as certificates. We describe a polynomial-time checker for testing whether a given CNF formula is equivalent or not to a given certifiable Decision-DNNF circuit. Finally, leveraging a modified version of the compiler D4 for generating certifiable Decision-DNNF circuits and an implementation of the checker, we present the results of an empirical evaluation that has been conducted for assessing how large are the certifiable Decision-DNNF circuits that can be generated in practice, and how much time is needed to compute and to check such circuits.

Introduction

Certifying the output of tools solving hard problems is necessary to ensure the correctness of the results they provide. Certification has proven to be fruitful to detect bugs in SAT-solvers. In SAT competitions, since 2013, it is mandatory that solvers output unsatisfiability certificates whenever the instance under consideration is classified as unsatisfiable. The widespread and quick adoption of certification techniques in the SAT-solving community has been facilitated by the fact that most SAT-solvers rely on the same architecture, namely CDCL, for which the underlying proof system is well-understood (Pipatsrisawat and Darwiche 2011) and for which a description such as DRAT (Wetzler, Heule, and Hunt 2014) – whose implementation follows closely the way the software works – is available.

The picture is quite different for tools tackling instances of more complex problems based on (possibly extended) CNF formulae such as QBF-solving, max SAT-solving or #SAT-solving. While many efforts have been invested into designing and understanding efficient and usable proof systems for QBF-solvers (Heule, Seidl, and Biere 2014; Beyersdorff, Chew, and Janota 2015; Beyersdorff et al. 2018) and max SAT-solvers (Bonet, Levy, and Manyà 2007; Bonet et al. 2018), there is no consensus on the format to be adopted for certifying the outputs generated by such solvers. Furthermore, from the practical side, there have been only few tries to adding a certification functionality to max SAT-solvers (Morgado and Marques-Silva 2011). To the best of our knowledge, today, no #SAT-solver is able to output a certificate that can be used to check in polynomial time that the result returned by the #SAT-solver is the correct one.

In order to achieve this goal, an approach consists in compiling the input CNF formula into a specific Boolean circuit. Indeed, it has been observed by Huang and Darwiche (Huang and Darwiche 2005) that the traces of DPLL-based #SAT-solvers on CNF formulae can be viewed as restricted forms of Boolean circuits known as Decision-DNNF circuits (Darwiche 2001; Oztok and Darwiche 2014) in the knowledge compilation community. The main advantage of such circuits is that they can later be used to analyze efficiently the Boolean function represented by the input CNF formula. Indeed, Decision-DNNF circuits support many tractable queries, such as model counting.

More sophisticated AI tasks leveraging propositional reasoning can also be targeted. Thus, in the growing body of work about explainable and robust AI (XAI) (see among others (Ribeiro, Singh, and Guestrin 2018; Leofante et al. 2018; Molnar, Casalicchio, and Bischl 2018; Shih, Darwiche, and Choi 2019; Guidotti et al. 2019; Miller 2019; Molnar 2019), recent works have shown how ML classifiers of various types can be mapped to CNF formulae encoding them in terms of input-output behaviours (see e.g., (Narodytska et al. 2018; Shih, Choi, and Darwiche 2019; Shi et al. 2020). Thanks to such mappings, XAI queries about classifiers (especially, verification queries) can be delegated to the corresponding propositional representations. Though those queries are typically NP-hard when the inputs are CNF formulae, many of them become tractable once compiled into Decision-DNNF circuits (Audemard, Koriche, and Marquis 2020). Obviously enough, within such an approach to XAI, the certification of the results furnished by the XAI system goes through the certification of the Decision-DNNF circuit that has been computed.

A first step in this direction is (Capelli 2019), where proof systems for #SAT that are suited for existing #SAT-solvers have been provided. The idea is to use the Decision-DNNF circuit corresponding to the trace of a DPLL-based #SAT-solver running on a CNF formula as a certificate witnessing the number of models of this CNF formula. While computing the number of models of the Decision-DNNF circuit can be done in polynomial time, no polynomial-time algorithm is available for deciding whether the input CNF formula actually represents the same Boolean function as the Decision-DNNF circuit (and it is unlikely that such an algorithm exists since the problem to be solved is coNP-complete). Using only the Decision-DNNF circuit makes it impossible to check whether the solver ran correctly on the input. To circumvent this issue, it has been proposed to add labels into the Decision-DNNF circuit. Those labels can be used (intuitively) to explain when the solver rejects a partial assignment. This approach is enough to make tractable the problem of checking whether the Decision-DNNF circuit that is generated is equivalent to the input CNF formula. Moreover, most DPLL-based #SAT-solvers could theoretically be modified to output such a certificate. In practice, however, this approach may need to be adapted to accommodate the specific optimizations of the solver such as its cache policy or oracle calls to a CDCL SAT-solver. In addition, no implementation of the proof systems presented in (Capelli 2019) has been provided yet.

In order to fill the gap, our main objective in this paper is to show how many top-down Decision-DNNF compilers can be tweaked so as to output certifiable Decision-DNNF circuits, that are mainly standard Decision-DNNF circuits decorated by additional pieces of information serving as certificates for addressing efficiently the verification issue. Among them are the state-of-the-art Decision-DNNF compilers, D4 (Lagniez and Marquis 2017a) and DSHARP (Muise et al. 2012). We start by defining the language of certifiable Decision-DNNF circuits and introduce a notion of equivalence between a CNF formula and a certifiable Decision-DNNF circuit, that we call syntactic equivalence. Syntactic equivalence is a restriction of logical equivalence (i.e., whenever a certifiable Decision-DNNF circuit and a CNF formula are syntactically equivalent, they are logically equivalent), and syntactic equivalence can be checked in polynomial time (which contrasts with logical equivalence, unless P = NP). This directly leads to a polynomial-time checker for testing whether a given CNF formula is equivalent or not to a given certifiable Decision-DNNF circuit. Then we explain how to modify a top-down Decision-DNNF compiler in order to generate certifiable Decision-DNNF circuits instead of Decision-DNNF circuits. Finally, leveraging CD4, a modified version of D4 for generating certifiable Decision-DNNF circuits, and an implementation of the checker, we present the results of an empirical evaluation that has been conducted for assessing in practice the sizes of the certifiable Decision-DNNF circuits, and the computation times needed to generate them and to check them.

The proofs of the propositions reported in the paper and a folder containing the code of CD4, the code of the checker, the benchmarks used in our experiments, and a spreadsheet containing detailed empirical results are available on www.cril.fr/kc/.

Preliminaries

Formulae and Boolean functions. We assume the reader familiar with Boolean functions and CNF formulae. We introduce a few notations that we will be used throughout the paper.

Given a finite set of variables X, $\{0,1\}^X$ denotes the set of all possible Boolean assignments to variables in X. Given a literal ℓ on variable x and a CNF formula F, $F[\ell]$ is the CNF formula over variables $\mathsf{var}(F) \setminus \{x\}$ obtained by removing every clause of F containing ℓ and by removing $\neg\ell$ from every remaining clause. It is easy to see that there is a one-to-one correspondence between the satisfying assignments of $F[\ell]$ and the set of satisfying assignments of F whose value on ℓ is 1. If $L = \{\ell_1, \ldots, \ell_k\}$ is a set of literals such that $\mathsf{var}(\ell_i) \neq \mathsf{var}(\ell_j)$ for $i \neq j$, then $F[L]$ is a short for $F[\ell_1]\ldots[\ell_k]$. Observe that $F[L]$ does not depend on the order of the elements in L. For example, if $F = (x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee \neg y)$, we have $F[\neg x] = y \vee \neg z$ and $F[\neg x, \neg y] = \neg z$. If F and F' are two CNF formulae, we write $F = F'$ if F and F' have exactly the same clauses after having removed duplicated clauses and tautological clauses (that is, clauses containing both a literal and its negation).

Given a CNF formula F on variables X and $Y \subseteq X$, the *connected component of F associated with Y*, denoted as $\mathsf{cc}(F, Y)$ is the smallest set S of clauses C of F such that S contains every clause of F such that $Y \cap \mathsf{var}(C) \neq \emptyset$ and for every clause C' of F, if $\mathsf{var}(C') \cap \mathsf{var}(S) \neq \emptyset$ then $C' \in S$. It is easy to see that one can construct $\mathsf{cc}(F, Y)$, starting from the set of clauses C of F such that $Y \cap \mathsf{var}(C) \neq \emptyset$ and iteratively adding to this set every clause of F having a non-empty intersection with it until no such clause remains.

If f and g are two Boolean functions, we write $f \models g$ if f entails g, that is, if $f \wedge \neg g$ is not satisfiable. DRAT (Wetzler, Heule, and Hunt 2014) is a proof system that has been specifically designed to certify the output of CDCL SAT-solvers. We refer the reader to (Wetzler, Heule, and Hunt 2014) for precise definitions as we will be using DRAT proofs as black boxes in this paper. We only need to know that a DRAT proof R can be seen as a list of clauses $R_1, \ldots, R_N$ such that for every i, $R_1 \ldots R_i \models_{\mathsf{DRAT}} R_{i+1}$, where $\models_{\mathsf{DRAT}}$ is a syntactic condition that can be checked in polynomial time and such that if $A \models_{\mathsf{DRAT}} B$ then $A \models B$. Given a CNF formula F and a DRAT proof $R = R_1, \ldots, R_N$, we write $F \models_{\mathsf{DRAT}} R$ if for every $i < N$, $F \cup R_1 \ldots R_i \models_{\mathsf{DRAT}} R_{i+1}$. Observe in particular that for every $i \leq N$, $F \models R_i$.

The language of Decision-DNNF circuits. The language of *Decision Decomposable Negation Normal Form circuits*, Decision-DNNF for short, is a class of restricted Boolean circuits used in knowledge compilation to represent Boolean functions. Formally, a *decision circuit* D on variables X is a DAG having one distinguished vertex v called the *output* of D (v is the unique vertex in D with out-degree zero). The

vertices of D with in-degree 0 are called the *sinks* and are labeled by constant 0 or 1. All vertices but the sinks are either $\wedge$-*gates*, labeled by $\wedge$, or *decision-gates* that are of in-degree 2 and labeled by a variable $x \in X$. For every decision-gate v labeled by variable x, the ingoing arcs (alias directed edges) e_0, e_1 of v are labeled by $\neg x$ and x, respectively. Given an arc e of D reaching a decision-gate, we denote by ℓ_e the literal labelling it.

Let D be a decision circuit. $\mathsf{var}(D)$ denotes the set of all variables appearing on the labels of the arcs of D (observe that $\mathsf{var}(D) \subseteq X$ but the inclusion may be proper). Given a gate v of D, D_v is the Decision-DNNF circuit whose output is v and gates are all gates w such that there is an (oriented) path from w to v. Given a gate v of D that is not a sink, we call the *inputs of* v the gates w such that (w, v) is an ingoing arc of v.

Given a $\wedge$-gate v of D with inputs v_1, v_2, v is said to be *decomposable* if $\mathsf{var}(D_{v_1}) \cap \mathsf{var}(D_{v_2}) = \emptyset$. The Decision-DNNF circuit D is said to be *read-once* if on every path P from a sink of D to the output of D, each variable of X appears at most once in the literals labelling the arcs of P.

A *Decision-DNNF* circuit is a decision circuit that is read-once and such that every $\wedge$-gate of D is decomposable. Let D be a Decision-DNNF circuit on variables X, P a path from a sink of D to the output of D, and $\tau \in \{0,1\}^X$. τ is said to be *compatible with* P if for every arc e of P going in a decision-gate, we have that $\tau(\ell_e) = 1$. We let $\mathsf{lit}(P)$ denote the set of all literals labelling the arcs it contains. Observe that the read-once property of decision-gates ensures that $\mathsf{lit}(P)$ is a consistent set of literals. The *size* of a Decision-DNNF circuit D, denoted as $|D|$, is the number of arcs of its underlying graph.

A Decision-DNNF circuit D on variables X defines a Boolean function on variables X defined inductively as follows. One defines first for every gate v of D the set of assignments $\tau \in \{0,1\}^X$ that are *accepted* by v. If v is a 1-sink, then it accepts every $\tau \in \{0,1\}^X$. If v is a 0-sink, then it accepts no $\tau \in \{0,1\}^X$. If v is a $\wedge$-gate, then v accepts $\tau \in \{0,1\}^X$ if and only if every input of v accepts τ. If v is a decision-gate, it accepts $\tau \in \{0,1\}^X$ if and only if there exists an input w of v such that the arc (w, v) is labeled by ℓ and $\tau(\ell) = 1$ and that w accepts τ. The Boolean function computed by D is the set of assignments accepted by its output. Observe that if D accepts τ then every path consistent with it from a sink to the output of D starts at a 1-sink of D. In this paper, we assume that the underlying graph of a Decision-DNNF circuit D is connected and that the output is the only gate of D that is of out-degree 0.

Top-down Decision-DNNF knowledge compilers. Top-down Decision-DNNF knowledge compilers are based on a natural generalization of the well-known DPLL algorithm for SAT-solving (Davis, Logemann, and Loveland 1962), so as to count models instead of deciding "only" whether a model exists (see e.g., (Birnbaum and Lozinskii 1999; Bacchus, Dalmao, and Pitassi 2003)). A top-down knowledge compiler works by recursively branching on a variable x until either a contradiction is reached or the formula is satisfied, relying on the simple observation that $F = (x \wedge F[x]) \vee (\neg x \wedge F[\neg x])$, which translates in terms of model counting into $\#F = \#F[x] + \#F[\neg x]$ when $\mathsf{var}(F[x]) = \mathsf{var}(F[\neg x])$. To avoid visiting all possible branches, the computed values are cached so that if the algorithm is recursively called on a subformula that has already been seen during computation, the number of models of this subformula is directly returned by the cache. An improvement of this algorithm has been proposed by Bayardo and Pehoushek (Bayardo Jr and Pehoushek 2000) who observed that if $F = F_1 \wedge F_2$ with $\mathsf{var}(F_1) \cap \mathsf{var}(F_2) = \emptyset$, then $\#F = \#F_1 \times \#F_2$ which allows to significantly reduce the number of recursive calls.

Darwiche and Huang established in (Huang and Darwiche 2005) that the trace of such an algorithm on a given CNF formula corresponds to a Decision-DNNF circuit where the operation of branching on a variable corresponds to a decision-gate and the operation of decomposing the formula into disjoint connected components corresponds to a $\wedge$-gate. Top-down Decision-DNNF knowledge compilers use at their core this algorithm whose pseudocode is presented in Algorithm 1. We will refer to this algorithm as a *generic top-down DPLL-based Decision-DNNFcompiler*, since it does not do anything else as propagating literals in the formula and checking for connected components.

Many top-down Decision-DNNF knowledge compilers, such as D4 or DSHARP, improve on this algorithm by taking advantage of a CDCL SAT-solver, used as an oracle for generating the compiled circuit. Leveraging a CDCL SAT-solver serves two purposes. The first purpose is to detect unsatisfiable branches as early as possible in order to avoid visiting them. The second purpose is to learn clauses that are entailed by the input CNF formula F. Indeed, when a conflict is reached in a CDCL SAT-solver, a clause formed by a minimal subset of literals generating this conflict is extracted. This clause is entailed by the CNF formula and it allows to detect further conflicts more quickly. In CDCL SAT-solvers, learnt clauses are used to trigger unit propagations that would have been missed when using the original clauses, only. The CDCL SAT-solver is configured in a way that each clause that has been learnt during this step is entailed by the original CNF formula F. The pseudocode of a *generic top-down CDCL-based Decision-DNNFcompiler* can be simply obtained by adding a couple of additional instructions into the pseudocode of a generic top-down DPLL-based Decision-DNNFcompiler (those instructions are framed in Algorithm 1). Observe that this pseudocode does not correspond to a specific top-down CDCL-based Decision-DNNFcompiler, but to a family of such compilers. In order to get specific compilers, some additional features should be specified (e.g., the branching variable heuristics and the cache management that are used).

Certifiable Decision-DNNF Circuits

It is coNP-complete to decide, given a CNF formula F and a Decision-DNNF circuit D, both on variables X, whether F and D define the same Boolean function (Capelli 2019). This can easily be seen by reducing it to the problem of deciding whether F is unsatisfiable, which boils down to

Algorithm 1: Pseudocode for Decision-DNNF top-down compilers. Specific instructions for certification in compilers based on a CDCL SAT-solver are framed.

Algorithm COMPILE(F)
  Data: A CNF formula F
  Result: A Decision-DNNF circuit associated with F
  cache $\leftarrow \emptyset$;
  return CONSTRUCT_GATE(F, $\emptyset$, $\emptyset$);

Algorithm CONSTRUCT_GATE(F, L [, R])
  Data:
  - **Input:** A CNF formula F
  - **Input/output:** A set L of literals
  - [**Input/output:** A set R of learnt clauses, entailed by F]

  Result: A Decision-DNNF-gate associated with $F[L]$

  if $F[L]$ *has no clause* **then return** *a new* 1*-sink*;
  if $F[L]$ *has an empty clause* **then return** *a new* 0*-sink*;
  [**if** $R[L]$ *has an empty clause* **then return** *a new* 0*-sink*;]

  [Call a SAT solver on F with literals in L blocked];
  [Let R' be the clauses learnt by this call];
  [$R \leftarrow R \cup R'$];

  [**if** *UNSAT(F)* **then return** *a new* 0*-sink*;]
  if cache$(F[L]) \neq$ nil **then return** cache$(F[L])$;
  $F' \leftarrow \{C \in F \mid C[L]$ is not satisfied$\}$;
  if $F' = F_1 \wedge \cdots \wedge F_k$ *with* $k \geq 2$ *and* $\mathsf{var}(F_i[L]) \cap \mathsf{var}(F_j[L]) = \emptyset$ **then**
    $v \leftarrow$ a new $\wedge$-gate;
    for $i \in \{1, \ldots, k\}$ **do**
      $w_i \leftarrow$ CONSTRUCT_GATE(F_i, L, R);
      connects v to w_i;
  else
    Choose $x \in \mathsf{var}(F[L])$;
    $v \leftarrow$ a new decision-gate on variable x;
    for $\ell \in \{x, \neg x\}$ **do**
      $w \leftarrow$ CONSTRUCT_GATE(F, $L \cup \{\ell\}$, R);
      connect v to w with label ℓ;
  cache$(F[L]) \leftarrow v$;
  return v

deciding whether F defines the same Boolean function as the Decision-DNNF circuit that is reduced to a single 0-sink. To be able to efficiently check whether the output of a knowledge compiler is correct, that is, to check whether the Boolean function represented by the output Decision-DNNF circuit is indeed equivalent to its input CNF formula, it is thus necessary to provide a certificate. In this paper, we consider Decision-DNNF circuits decorated with annotations that are used to efficiently check the equivalences with the CNF formulae represented by the Decision-DNNF circuits. They are called certifiable Decision-DNNF circuits.

The language of certifiable Decision-DNNF is closely related to the language of "certified" Decision-DNNF circuits, as introduced in (Capelli 2019). The "certified" Decision-DNNF circuits from (Capelli 2019) only contain certificates for 0-sinks, in the sense that each input labeled by 0 should also be labeled by a clause of the original CNF formula F – or, more generally, by a clause C that is entailed by F, with a proof of such a fact. Moreover, each path from this 0-sink to the output of D has to violate C. Intuitively, C is the clause that is responsible for the conflict when the solver reached it. For this reason, the syntactic cache management that is used in implemented compilers like D4 and DSHARP must be disabled if one wants to take advantage of them to compute "certified" Decision-DNNF circuits. Indeed, it is possible that two clauses, for example $C_1 = x \vee y \vee z$ and $C_0 = x \vee y \vee \neg z$, become equal under two distinct partial assignments. In this case, we could have a cache hit in the solver after having set z to 0 then to 1, but the conflict generated by later assigning x and y to 0 would not be raised by the same clause, depending on the value of z. Thus, the approach based on "certified" Decision-DNNF circuits as described in (Capelli 2019) is not suited for compilers using such a cache management. This motivated the introduction of certifiable Decision-DNNF circuits that are better suited to certify the output of top-down CDCL-based Decision-DNNF compilers.

Formally, a *certifiable Decision-DNNF* circuit (D, e, R) is a Decision-DNNF circuit D together with a list of clauses R and a labelling e associating with every gate v of D, but its output, an arc $e(v)$ going out of v. $e(v) = (v, w)$ is called the *canonical arc of* v and w is the *canonical father of* v.

Let $e = (v, w)$ be an arc of D. We define F/e as follows: if w is a decision-gate, then $F/e = F[\ell_e]$; if w is a $\wedge$-gate, then $F/e = \mathsf{cc}(F, \mathsf{var}(D_v))$.

Let F be a CNF formula on variables X and v a gate of D. We define F_v and $\mathsf{lit}(v)$ inductively as follows: if v is the output of D, then $F_v = F$ and $\mathsf{lit}(v) = \emptyset$. Otherwise, let $(v, w) = e(v)$. We define $F_v = F_w/e(v)$ and $\mathsf{lit}(v) = \mathsf{lit}(w) \cup \ell$ if w is a decision-gate and ℓ is the literal labelling $e(v)$. Otherwise, $\mathsf{lit}(v) = \mathsf{lit}(w)$.

We can then extend this definition to any path P from a node v to the output of D: if P is empty, then we let $F/P = F$. Otherwise, let $e = (v, w)$ be the first arc on P and P' be the path from w to the output of D obtained by removing e from P. F/P is defined as $(F/P')/e$.

As explained previously, each gate of the Decision-DNNF circuit produced by a top-down compiler corresponds to a recursive call of Algorithm 1. In a certifiable Decision-DNNF circuit, the canonical arc $e(v)$ of a gate v intuitively corresponds to the first arc that was connected to v after its creation. Moreover, R corresponds to the clauses that have been learnt during the compilation through calls to the CDCL SAT-oracle, and F_v corresponds to the state the solver was in when v has been created.

This motivates the following definition of syntactic equivalence. A certifiable Decision-DNNF circuit (D, e, R) is *syntactically equivalent to a CNF formula* F, denoted as $F \equiv_s (D, e, R)$, if $\mathsf{var}(D) \subseteq \mathsf{var}(F)$ and the following conditions are met:

(i) **Learnt clauses entailment**: $F \models R$.

(ii) **Satisfiable gates:** if $R \neq \emptyset$, then for every gate v that is not a 0-sink of D, D_v is satisfiable.

(iii) **DNNF entailment**: $D \models F$.

(iv) **Cache consistency:** for every gate v of D that is not a sink and arc $e = (v, w)$ of D, $F_w/e = F_v$.

(v) **0-sinks consistency:** for every 0-sink v and arc $e = (v, w)$, either F_w/e contains the empty clause or w is

a decision-gate and $R[\mathsf{lit}(w) \cup \ell_e]$ contains the empty clause.

Clearly enough, syntactic equivalence is a restriction of equivalence in the sense that whenever a certifiable Decision-DNNF circuit is syntactically equivalent to a CNF formula, it is logically equivalent to it, but the converse does not hold. Indeed, consider the CNF formula $F = (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$ and a certifiable Decision-DNNF circuit (D, e, R) where D is reduced to a single decision-gate w on variable x connected only to the 0-sink v (so that e is reduced to $\{(v, w)\}$) and R is empty (there are no learnt clauses). (D, e, R) is a certifiable Decision-DNNF circuit logically equivalent to F but not syntactically equivalent to F as Condition v is not satisfied.

Example of certifiable Decision-DNNF circuit. To illustrate the notion of certifiable Decision-DNNF circuit and the previous definition of syntactic equivalence, we now provide a full example. Let us consider the following CNF formula:

$$\begin{aligned} F = &(\neg x \vee \neg y) \wedge (x \vee z) \wedge \\ &(x \vee a \vee w) \wedge (x \vee \neg a \vee w) \wedge \\ &(\neg x \vee y \vee a \vee w) \wedge (\neg x \vee y \vee \neg a \vee w) \end{aligned}$$

Let us also consider the Decision-DNNF circuit D represented on Figure 1. Some arcs of this Decision-DNNF circuit have been represented in bold. They correspond to the canonical arcs: a bold arc going out of a vertex v is the canonical arc $e(v)$ of v. $R = \{\neg x \vee y \vee w\}$ is the set of learnt clauses.

We claim that the certifiable Decision-DNNF circuit (D, e, R) is syntactically equivalent to F. It is clear that Condition (i) is satisfied as $F \models R$ holds (it can be shown by applying the resolution rule to the last two clauses of F). Condition (ii) can also be readily verified. Condition (iii) can be checked by tested that for every clause C of F, the circuit D conditioned by the negation of C is not satisfiable. For example, focusing on the clause $x \vee z$ of F, the path of D associated with $\neg x \wedge \neg z$ in D starts from a 0-sink.

We now turn to checking Condition (iv) and (v). To this end, we observe the following:

- $F_{v_1} =_{def} F[x] = \neg y \wedge (y \vee a \vee w) \wedge (y \vee \neg a \vee w)$,
- $F_{v_2} =_{def} F[\neg x] = (a \vee w) \wedge (\neg a \vee w) \wedge z$,
- $F_{v_3} =_{def} (F_{v_1})[\neg y] = (a \vee w) \wedge (\neg a \vee w)$,
- $F_{v_4} = z$, as $\mathsf{var}(D_{v_4}) = \{z\}$ and the connected component corresponding to z in F_{v_3} consists only of the unit clause z.

Condition (iv) is trivially verified at nodes having exactly one outgoing arc. We just have to check it at gate v_3. That is, we have to verify that $F_{v_3} = F_{v_2}/e$ where $e = (v_2, v_3)$. By definition, F_{v_2}/e consists of all clauses of F_{v_2} that are in the connected component of $\mathsf{var}(D_{v_3}) = \{w\}$. Thus, $F_{v_2}/e = (a \vee w) \wedge (\neg a \vee w)$ which is indeed F_{v_3} from what precedes.

Finally, we have to check Condition (v) for every 0-sink of D. We start with the arc f from the 0-sink going in v_1. We have to show that F_{v_1}/f contains the empty clause. By definition, $F_{v_1}/f = F_{v_1}[y]$, which indeed contains the empty clause. A similar check can be done with the arc going in v_4.

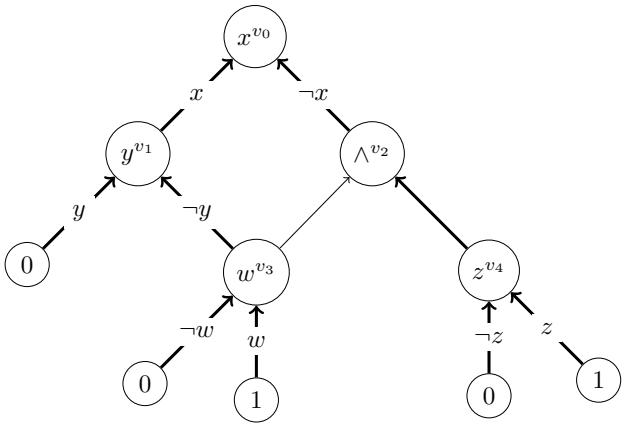


Figure 1: A certifiable Decision-DNNF circuit.

The case of the arc g between the 0-sink and v_3 is more interesting. Indeed, by definition $F_{v_3}/g = F_{v_3}[\neg w] = a \wedge \neg a$. It does not contain the empty clause. However, it can be checked that $\mathsf{lit}(v_3) = \{x, \neg y\}$ and $R[\mathsf{lit}(v_3) \cup \neg w] = R[x, \neg y, \neg w]$ contains the empty clause, that is, Condition (v) holds. Observe that in this case, the learnt clause helped to detect a conflict that was not direct in F_{v_3}/g. Indeed, if the canonical path to v_3 was $P = \{v_3, v_2, v_0\}$, then since $\mathsf{lit}(P) = \{\neg x, \neg w\}$, we would have $R[\neg x, \neg w] = \emptyset$, which does not contain the empty clause. In this case, Condition (v) would not be verified. This example highlights the importance of registering canonical arcs in the certifiable Decision-DNNF circuit to check for unit propagations entailed by learnt clauses.

Syntactic equivalence of a certifiable Decision-DNNF circuit with a CNF formula. We can now describe easily how to implement a verification algorithm CHECKER for deciding whether or not a given certifiable Decision-DNNF circuit D is syntactically equivalent to a given CNF formula F. The verification algorithm just consists in verifying all the conditions for syntactic equivalence in a successive way. Our actual implementation, presented in the following, contains some optimizations to better scale up for large CNF formulae and certifiable Decision-DNNF circuits. A key property is that the verification step is tractable:

Theorem 1 *Let F be a CNF formula and (D, e, R) be a certifiable Decision-DNNF circuit on variables X. Under the assumption that $F \models R$, we can check whether $F \equiv_s (D, e, R)$ in polynomial time.*

The modified version of Algorithm 1 outputs a set of clauses R that have been learnt via oracle calls to a CDCL SAT-solver. R is such that $F \models_{\mathsf{DRAT}} R$ (Wetzler, Heule, and Hunt 2014) (meaning that every clause from R has a DRAT proof from F) and this implication can be checked in polynomial time.

Moreover, the syntactic equivalence of a CNF formula and a certifiable Decision-DNNF circuit ensures that they are logically equivalent:

Theorem 2 *Let F be a CNF formula and (D, e, R) be a certifiable Decision-DNNF circuit such that $F \equiv_s (D, e, R)$. It holds that $F \equiv D$.*

The intuition behind Theorem 2 is the following. Conditions (iv) and (v) check that the hypothesis made by the solver when caching, decomposing or deciding for a conflict were legit, the state of the solver when it took the decision corresponding to F_v. Condition (iii) ensures one way of the equivalence. It is tractable even without the additional decoration of certifiable Decision-DNNF as it boils down to checking $D \models C$ for every clause C of F, and this query (known as clause entailment) is tractable for the Decision-DNNF language (Darwiche and Marquis 2002). Finally, Conditions (ii) and (i) ensure that conflicts derived from learnt clauses of R are also conflicts of F.

Condition (ii) may seem surprising and unnecessary but it is actually essential to ensure that, when checking for Condition (v) if $R[\mathsf{lit}(v)]/e$ contains a conflict for some gate v of D, then this conflict is also a logical consequence of F_v, that is the only formula for which the cache consistency has been checked. The potentially incorrect interaction between the caching policy and the use of learnt clauses was already known to be problematic in parallel model counter (Burchard, Schubert, and Becker 2015) and in CACHET (Sang et al. 2004) where it has been resolved thanks to a method known as *sibling pruning*, that consists in cleaning the cache of unsatisfiable gates and that corresponds to Condition (ii). In Algorithm 1, this condition is guaranteed by the fact that oracle calls to CDCL SAT-solvers are done at each recursion, ensuring that unsatisfiable branches are never explored. Note that for DPLL-based compilers, there are no conflicts involving learnt clauses, so that R is empty and Condition (ii) also holds.

Finally, observe that the labelling e in the (D, e, R) triple is necessary to certify the circuit as it allows to recover the partial assignment L that the solver was considering when creating a new gate. It is crucial as it may be that some unit propagation triggered by learnt clauses under partial assignment L is not triggered under another partial assignment L', even if $F[L] = F[L']$. By recording the canonical father of every gate of D (but its output), we can reconstruct L during the certification and thus certify the unit propagations that were triggered by learnt clauses at this point.

Turning top-down Decision-DNNF compilers into certifiable Decision-DNNF compilers. In this section, we explain how top-down knowledge compilers of the family presented in Algorithm 1 can be tweaked so as to generate certifiable Decision-DNNF circuits that are syntactically equivalent to the input CNF formula.

The only change that has to be performed is that each time a new gate w is created by Algorithm 1, we define $e(w)$ to be the first arc going out of w that is created by the solver. To do so, we only have to modify the CONSTRUCT_GATE procedure in such a way that instead of returning a gate, it returns a gate and a Boolean flag telling whether the gate has been obtained from a cache hit or by constructing it from scratch. Then, in Lines 18 and 24 of Algorithm 1, if one detects that the returned gate w has not resulted from a cache hit, then one registers $e(w)$ to be (w, v).

Accordingly, the DAG D generated by CD4 is the same as the one generated by D4, which shows that certification does not change the size of the Decision-DNNF circuit that is produced: only the certification labellings may increase its size (of course, they can be left aside afterwards, when leveraging the circuit for reasoning purposes).

Finally, the COMPILE procedure returns (D, e, R) where D is the Decision-DNNF circuit that has been constructed by the algorithm, where e is the labelling that has been defined above, and R is the set of all clauses learnt during the compilation. For any instance of Algorithm 1 that is not using a CDCL SAT-solver, R is empty so that $(D, e, \emptyset)$ is returned.

Interestingly, compared to the pseudocode of the generic compiler targeting Decision-DNNF as given by Algorithm 1, only a few additional instructions must be inserted to derive certifiable Decision-DNNF circuits. We denote by C_COMP ("certifiable compiler") the generic compiler given by Algorithm 1 when expanded with those instructions. The correctness of C_COMP is established by the next theorem:

Theorem 3 *Let F be a CNF formula and (D, e, R) be the output of* C_COMP *on input F. It holds that (D, e, R) is a certifiable Decision-DNNF circuit such that $F \equiv_s (D, e, R)$ and $F \models_{\mathsf{DRAT}} R$.*

From Theorem 3, 1, 2 and the fact that $F \models_{\mathsf{DRAT}} R$ can be checked in polynomial time, one gets that the output of C_COMP can be verified in polynomial time.

Experimental Results

Empirical setting. We have implemented the CD4 compiler, obtained by tweaking the state-of-the-art Decision-DNNF compiler D4 (Lagniez and Marquis 2017a), following the approach described in the previous sections, so as to generate certifiable Decision-DNNF circuits. CD4 is thus the instance of C_COMP associated with D4. We have also implemented the checker CHECKER described previously.

The objective of our experiments was twofold. A first goal was to assess the difficulty of computing certifiable Decision-DNNF circuits in comparison to (unconstrained) Decision-DNNF circuits. From the theory side, it can be easily established that the full language of Decision-DNNF circuits is strictly more succinct than the language of certifiable Decision-DNNF circuits unless NP = coNP. Indeed, any unsatisfiable CNF formula, whatever its size, is equivalent to the Decision-DNNF circuit that reduces to a single 0-sink, but there is no polynomial-size certificates for unsatisfiable CNF formulae unless NP = coNP. In the worst case, the size of a certifiable Decision-DNNF circuit for a given unsatisfiable CNF formula must thus be super-polynomial in the size of the CNF formula, hence in practice, arbitrarily larger than the size of the Decision-DNNF circuit that reduces to a single 0-sink. For this reason, it is important to

evaluate from the practical side the extra effort needed to derive certifiable Decision-DNNF circuits in comparison to the one required by the generation of (unconstrained) Decision-DNNF circuits.

A second goal was to assess the difficulty of computing a *certified* Decision-DNNF circuit associated with a given CNF formula. To do the job, the computation time aggregates the one needed to derive first a certifiable Decision-DNNF circuit for the CNF formula, and then to verify that this certifiable Decision-DNNF circuit is equivalent to the CNF formula, using the above-mentioned checker. Indeed, though the checker runs in time polynomial in its input size, the sizes of the certifiable Decision-DNNF circuits and the sizes of the DRAT proofs that are generated can be very large in comparison to the one of the CNF formula, rendering prohibitive in practice the time required by the verification step. Experiments are required to determine the extent to which the verification step can be done.

In our experiments, we have considered 703 CNF instances from the SATLIB[1] and other repositories (for instance, the benchmarks from the BN family (Bayesian networks) come from http://reasoning.cs.ucla.edu/ace/). They are gathered into 8 data sets, as follows: BN (192), BMC (Bounded Model Checking) (18), Circuit (41), Configuration (35), Handmade (58), Planning (248), Random (104), Qif (7) (Quantitative Information Flow analysis - security).

All the experiments have been conducted on a cluster equipped with quadcore bi-processors Intel XEON E5-5637 v4 (3.5 GHz) and 128 GiB of memory. The kernel used was CentOS 7, Linux version 3.10.0-514.16.1.el7.x86_64. The compiler used was gcc version 5.3.1. Hyperthreading was disabled, and no cache share between cores was allowed. A time-out of 1h for the generation of certifiable Decision-DNNF circuits plus 1h for the verification step has been considered per instance. A memory-out of 7.6 GiB has been considered per instance.

Results. We have first measured the number of instances (out of 703) for which a certifiable Decision-DNNF circuit has been computed by CD4 in due time given the allocated ressources and compared it with the number of instances for which a Decision-DNNF circuit has been computed by D4 using the same time and memory ressources. It turns out that 584 instances over 703 have been solved by D4, and that among them, CD4 has been able to solve 580 instances, hence more than 99% of the instances solved by D4.

Then we focused on the generation of certified Decision-DNNF circuits (taking account for the verification time within a time limit of 1h). For 69 out of the 580 certifiable Decision-DNNF circuits generated by CD4, the verification of the DRAT proofs involved in the certificates using the DRAT-trim proof checker[2] (Wetzler, Heule, and Hunt 2014) crashed with a segmentation fault, and for 42 additional instances a time-out has been reached before the checker terminated. Thus, 469 circuits out of 580 have been certified. Assuming that the DRAT proofs are ok for the 69 instances for which the DRAT-trim proof checker crashed, a proportion of more than 92% of the Decision-DNNF circuits that have been computed would have been certified.

The resulting values clearly show that in practice, the extra effort needed by the certification requirement does not drastically reduce the set of instances that can be addressed within a reasonable time. This observation highly contrasts with what is predicted from the theory side, suggesting that the worst scenario is not encountered very often in practice.

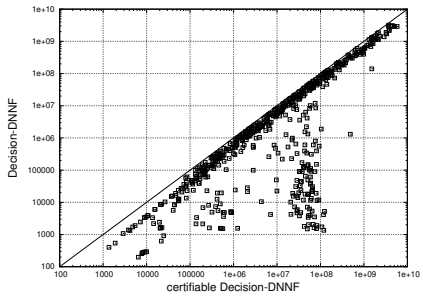
We have performed more fine-grained measurements at the instance level, in order to determine for each CNF instance (out of 580) the size of the (possibly decorated) circuits that have been generated by CD4 and by D4. We have considered a new output format for representing Decision-DNNF circuits and certifiable Decision-DNNF circuits (see www.cril.fr/kc/ for details); unlike the previous format used (which was suited to the representation of (general) DNNF circuits (Darwiche 2001)), decision-gates are now represented natively and literals obtained via unit propagation once a decision has been made label the corresponding arc). For the sake of homogeneity, sizes are measured in bytes (the size of a certifiable Decision-DNNF circuit includes the arcs that are generated, the extra-information labelling the arcs, as well as the sizes of the DRAT proofs of the clauses that have been learnt). We also measured for each CNF instance (out of 469) the time needed to get a certified Decision-DNNF circuit equivalent to it, by generating first a certifiable Decision-DNNF circuit and adding to this generation time the time required by the checker to ensure that this certifiable Decision-DNNF circuit is a certified Decision-DNNF circuit equivalent to the CNF formula as input. Then we were able to compare it with the the time needed by D4 to derive a Decision-DNNF circuit equivalent to the input CNF formula.

Interestingly, our approach to certification was able to handle large benchmarks, like the `blockmap_22_03.net` instance from the BN family (the formula has 119003 variables and 247486 clauses). Indeed, a certifiable Decision-DNNF circuit for it has been computed in 275s, and its size is approximately twice the size of the corresponding Decision-DNNF circuit.

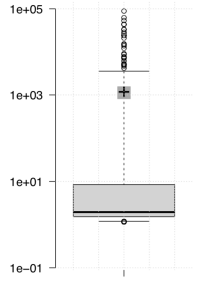
Our experiments have also shown that the time needed to certify a certifiable Decision-DNNF circuit in light of the CNF formula used to generate it typically depends on the size of the circuit, but not solely of it. For instance, the benchmark `lang12` from the Handmade family led to a certifiable Decision-DNNF circuit represented using 1518068287 bytes and the certification of the latter using the checker required 263s (242s being spent in verifying the DRAT part of the certificate). This contrasts with the instance `emptyroom_d20_g10_corners_p_t8` from the Planning domain which has been associated with a certifiable Decision-DNNF circuit that is more than twice larger (3351189294 bytes), but has been certified in 141s only (using less than 2s in the verification of the DRAT part of the certificate). This can be explained by the fact that `emptyroom_d20_g10_corners_p_t8` contains less than half clauses as `lang12`.

[1] www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html

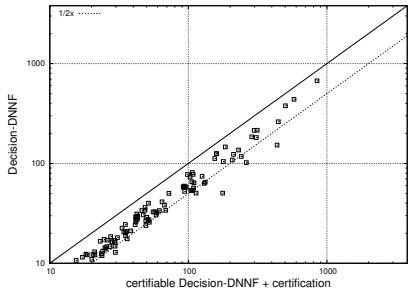
[2] The DRAT-trim proof checker is available at https://github.com/marijnheule/drat-trim.



(a) certifiable Decision-DNNF vs. Decision-DNNF: sizes of the compiled forms



(b) Distribution of the size ratios $\frac{s_c}{s}$



(c) certified Decision-DNNF vs. Decision-DNNF: computation times

Figure 2: Figure 2a compares the sizes of the compiled forms that are obtained when certifiable Decision-DNNF and Decision-DNNF are targeted. Figures 2b and 2c give respectively the distribution of the size ratios and a comparison of the times needed to get certified Decision-DNNF circuits with the times needed to get Decision-DNNF circuits.

Some results are synthesized in the scatter plot depicted at Figure 2a. Each dot in the figure corresponds to one of the instances. The y-axis indicates the size (in bytes) of the Decision-DNNF circuit computed by D4, while the x-axis gives the size (in bytes) of the certifiable Decision-DNNF circuit computed by CD4. Logarithmic scales are used for the two axes in the figure.

For each of the 580 instances for which both D4 and CD4 terminated in due time, we have also computed the size ratio $\frac{s_c}{s}$, where s_c (resp. s) is the size of the certifiable Decision-DNNF circuit (resp. Decision-DNNF circuit) generated by CD4 (resp. D4). Figure 2b gives a boxplot picture that is helpful to figure out how much spread the distribution of the size ratios is. Though the median size ratio is quite small (1.94) and the upper whisker is reasonable enough (19.31), the distribution has a number of outliers, corresponding to values that are really far from the values usually encountered (the maximum size ratio is equal to 88409).

Accordingly, it turns out that the additional space needed to represent the information used for the certification task remains reasonable enough for many instances (as reflected by the fact that the certifiable Decision-DNNF compilations succeeded most of the time – hence without leading to an error-of-memory).

Finally, Figure 2c is similar to Figure 2a, but it focuses on the times needed to derive certified Decision-DNNF circuits in comparison to the times needed to derive Decision-DNNF circuits. Thus, the x-axis represents the cumulated time (in seconds) needed to get a certified Decision-DNNF circuit of the instance, by using CD4 first to generate a certifiable Decision-DNNF circuit and the checker next to verify that it is equivalent to the instance. This figure shows that when the certification step succeeds, this extra time remains reasonable enough. Thus, for a large majority of instances out of 469, the cumulated time required first by CD4 to generate a certifiable Decision-DNNF circuit, then by the checker to verify it, is less than twice the time required by D4 to generate a Decision-DNNF circuit (the median cumulated time is equal to 1.69s). In the global verification time, the part of the time spent for verifying the DRAT proofs highly depends on the instance under consideration. For instance, as already mentioned, it took almost all the verification time for the benchmark `lang12`. Contrastingly, though the global verification time used for the instance `sat-grid-pbl-0030` from the BN family was significant (159s), the amount of it taken by the verification of the DRAT proofs was negligible (less than 1s).

Conclusion

In this paper, we have introduced the language of certifiable Decision-DNNF circuits for the certification purpose. A key feature of certifiable Decision-DNNF circuits is that a strong notion of equivalence to CNF formulae can be tested in polynomial time (while the problem is coNP-complete when Decision-DNNF circuits are considered instead).

We have explained how compilers from a general family of top-down Decision-DNNF compilers (including D4 and DSHARP) can be modified so as to output certifiable Decision-DNNF circuits. Using a modified version of D4 for generating certifiable Decision-DNNF circuits and an implementation of the checker, we have presented the results of an empirical evaluation showing that the sizes of the certifiable Decision-DNNF circuits, and the times needed to compute them and to check them, remain in general small enough to ensure the feasibility of the approach from the practical side.

A final remark is that since the model counts of certifiable Decision-DNNF circuits can be computed in polynomial time, one can obviously take advantage of certifiable Decision-DNNF circuits to derive certified numbers of models for CNF formulae.

This paper opens a number of perspectives for further research. One of them will consist in analyzing the language of certifiable Decision-DNNF circuits in a more systematic way so as to determine the queries and transformations it offers and to insert it into the knowledge compilation map.

It would be also valuable to compare in practice the performances of our approach to certification with other techniques. For instance, when dealing with CNF instances which are known to have (relatively) "few models", an ap-

proach to certifying their numbers consists in enumerating those models – a polynomial-time model enumeration algorithm exists for Decision-DNNF circuits – and once the enumeration is done, to test that every model of the CNF instance has been enumerated – this requires to consider a DRAT proof of the fact that the formula entails the disjunction of all the models found. Another approach would consist in building a Miter to test the equivalence of the input CNF formula and the output Decision-DNNF circuit. Checking the equivalence of both representations would require to leverage a SAT-oracle. A valuable feature of this approach is that it may be less space demanding than computing a certifiable Decision-DNNF circuit. On the other hand, the equivalence check boils down to solving an instance of a coNP-complete problem, so that one cannot get any guarantee on the execution time of the checker. This contrasts with our approach where the complexity of the certification step is directly related to the size of the certificate.

Finally, since the performances of D4 over many CNF instances are typically boosted when those instances have been first preprocessed using pmc (Lagniez and Marquis 2017b) (or B+E (Lagniez, Lonca, and Marquis 2020) when D4 is used as a model counter), it would be useful to develop and evaluate certification techniques for such preprocessors. This would permit to take advantage of them upstream to CD4, while maintaining the certification requirement.

Acknowledgments

This work has been partly supported by the PING/ACK project (ANR-18-CE40-0011) and the AI Chair EXPEKCTATION (ANR-19-CHIA-0005-01) from the French National Agency for Research. It was also supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215

References

Audemard, G.; Koriche, F.; and Marquis, P. 2020. On Tractable XAI Queries based on Compiled Representations. In *Proc. of KR'20*, 838–849.

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. Algorithms and Complexity Results for #SAT and Bayesian Inference. In *Proc. of FOCS'03*, 340–351.

Bayardo Jr, R. J.; and Pehoushek, J. D. 2000. Counting models using connected components. In *Proc. of AAAI'00*, 157–162.

Beyersdorff, O.; Chew, L.; and Janota, M. 2015. Proof complexity of resolution-based QBF calculi. In *Proc. of STACS'15*, volume 30, 76–89.

Beyersdorff, O.; Chew, L.; Mahajan, M.; and Shukla, A. 2018. Understanding cutting planes for QBFs. *Information and Computation* 262: 141–161.

Birnbaum, E.; and Lozinskii, E. L. 1999. The Good Old Davis-Putnam Procedure Helps Counting Models. *Journal of Artificial Intelligence Research* 10: 457–477.

Bonet, M. L.; Buss, S.; Ignatiev, A.; Marques-Silva, J.; and Morgado, A. 2018. MaxSAT resolution with the dual rail encoding. In *Proc. of AAAI'18*, 6565–6572.

Bonet, M. L.; Levy, J.; and Manyà, F. 2007. Resolution for Max-SAT. *Artificial Intelligence* 171(8–9): 606–618.

Burchard, J.; Schubert, T.; and Becker, B. 2015. Laissez-Faire Caching for Parallel #SAT Solving. In Heule, M.; and Weaver, S., eds., *Proc. of SAT'15*, 46–61.

Capelli, F. 2019. Knowledge compilation languages as proof systems. In *Proc. of SAT'19*, 90–99.

Darwiche, A. 2001. Decomposable negation normal form. *Journal of the ACM* 48(4): 608–647.

Darwiche, A.; and Marquis, P. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* 17: 229–264.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A Machine Program for Theorem-proving. *Communications of the ACM* 5(7): 394–397.

Guidotti, R.; Monreale, A.; Ruggieri, S.; Turini, F.; Giannotti, F.; and Pedreschi, D. 2019. A Survey of Methods for Explaining Black Box Models. *ACM Computing Surveys* 51(5): 93:1–93:42.

Heule, M. J.; Seidl, M.; and Biere, A. 2014. A unified proof system for QBF preprocessing. In *Proc. of IJCAR'14*, 91–106.

Huang, J.; and Darwiche, A. 2005. DPLL with a Trace: From SAT to Knowledge Compilation. In *Proc. of IJCAI'05*, 156–162.

Lagniez, J.; Lonca, E.; and Marquis, P. 2020. Definability for model counting. *Artificial Intelligence* 281: 103229.

Lagniez, J.; and Marquis, P. 2017a. An Improved Decision-DNNF Compiler. In *Proc. of IJCAI'17*, 667–673.

Lagniez, J.; and Marquis, P. 2017b. On Preprocessing Techniques and Their Impact on Propositional Model Counting. *Journal of Automated Reasoning* 58(4): 413–481.

Leofante, F.; Narodytska, N.; Pulina, L.; and Tacchella, A. 2018. Automated Verification of Neural Networks: Advances, Challenges and Perspectives. *CoRR* abs/1805.09938.

Miller, T. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence* 267: 1–38.

Molnar, C. 2019. *Interpretable Machine Learning - A Guide for Making Black Box Models Explainable*. Leanpub.

Molnar, C.; Casalicchio, G.; and Bischl, B. 2018. iml: An R package for Interpretable Machine Learning. *Journal of Open Source Software* 3(26): 786.

Morgado, A.; and Marques-Silva, J. 2011. On validating Boolean optimizers. In *Proc. of ICTAI'11*, 924–926.

Muise, C.; McIlraith, S.; Beck, J.; and Hsu, E. 2012. Dsharp: Fast d-DNNF Compilation with sharpSAT. In *Proc. of AI'12*, 356–361.

Narodytska, N.; Kasiviswanathan, S. P.; Ryzhyk, L.; Sagiv, M.; and Walsh, T. 2018. Verifying Properties of Binarized Deep Neural Networks. In *Proc. of AAAI'18*, 6615–6624.

Oztok, U.; and Darwiche, A. 2014. On Compiling CNF into Decision-DNNF. In *Proc. of CP'14*, 42–57.

Pipatsrisawat, K.; and Darwiche, A. 2011. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence* 175(2): 512–525.

Ribeiro, M. T.; Singh, S.; and Guestrin, C. 2018. Anchors: High-Precision Model-Agnostic Explanations. In *Proc. of AAAI'18*, 1527–1535.

Sang, T.; Bacchus, F.; Beame, P.; Kautz, H. A.; and Pitassi, T. 2004. Combining Component Caching and Clause Learning for Effective Model Counting. *Proc. of SAT'04* .

Shi, W.; Shih, A.; Darwiche, A.; and Choi, A. 2020. On Tractable Representations of Binary Neural Networks. In *Proc. of KR'20*, 882–892.

Shih, A.; Choi, A.; and Darwiche, A. 2019. Compiling Bayesian Networks into Decision Graphs. In *Proc. of AAAI'19*, 7966–7974.

Shih, A.; Darwiche, A.; and Choi, A. 2019. Verifying Binarized Neural Networks by Angluin-Style Learning. In *Proc. of SAT'19*, 354–370.

Wetzler, N.; Heule, M. J.; and Hunt, W. A. 2014. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *Proc. of SAT'14*, 422–429.